Mohamad Alsabbagh

Tarjan Algorithm

Java Implementation

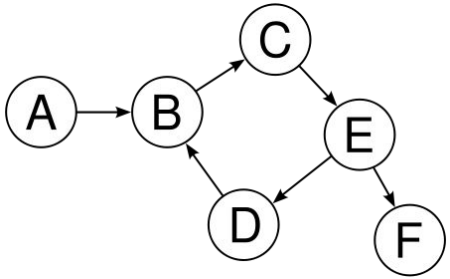
Department of Electrical Engineering and Computer Science York University, Toronto

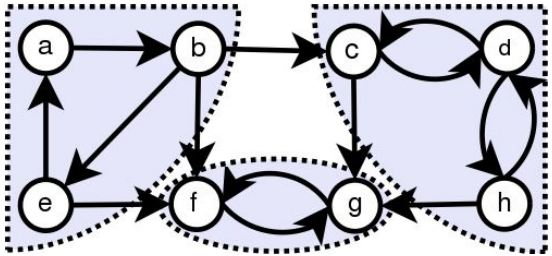
November 5, 2015

Problems Tarjan’s algorithms solve

Tarjan's Algorithms solve three related problems relevant to model checking. Given a state graph;

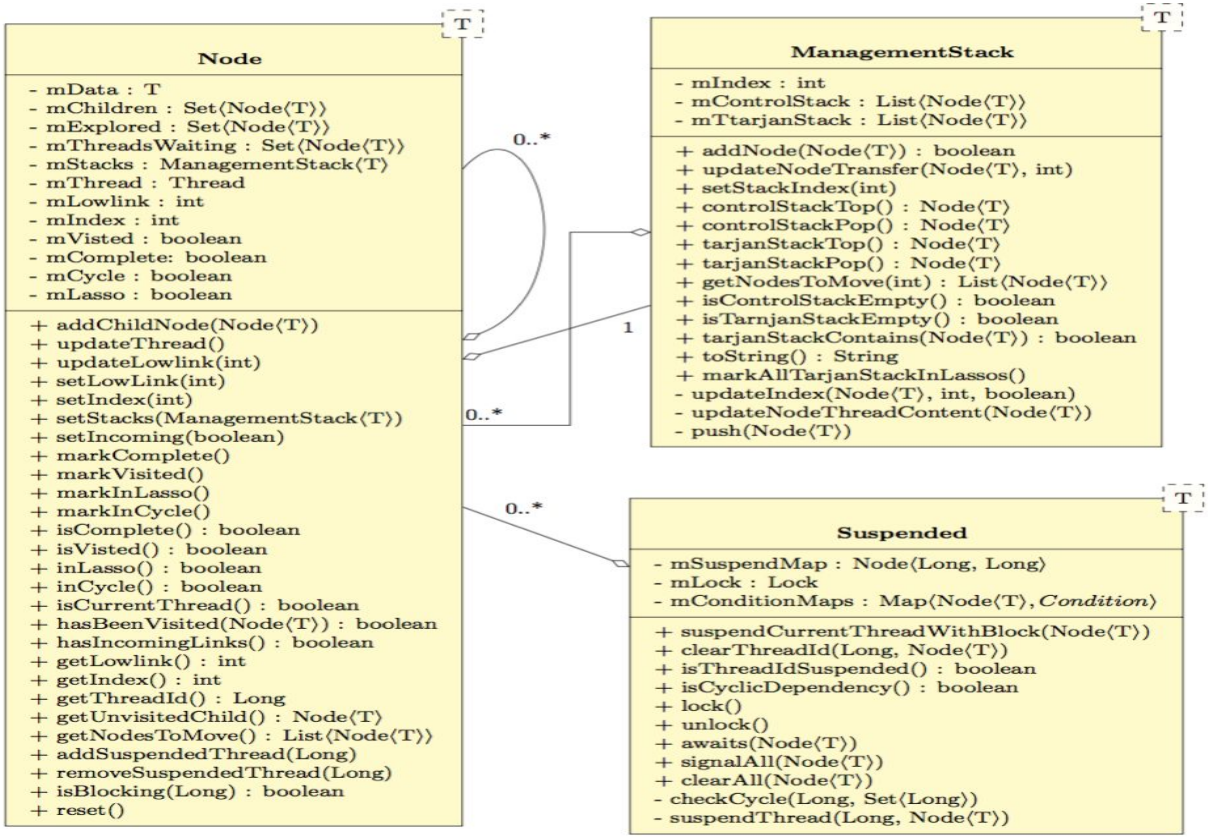
- Find its Strongly Connected Components (SCCs)
- Identify which nodes are in a loop
- Locate which nodes are in a lasso





Data Structures

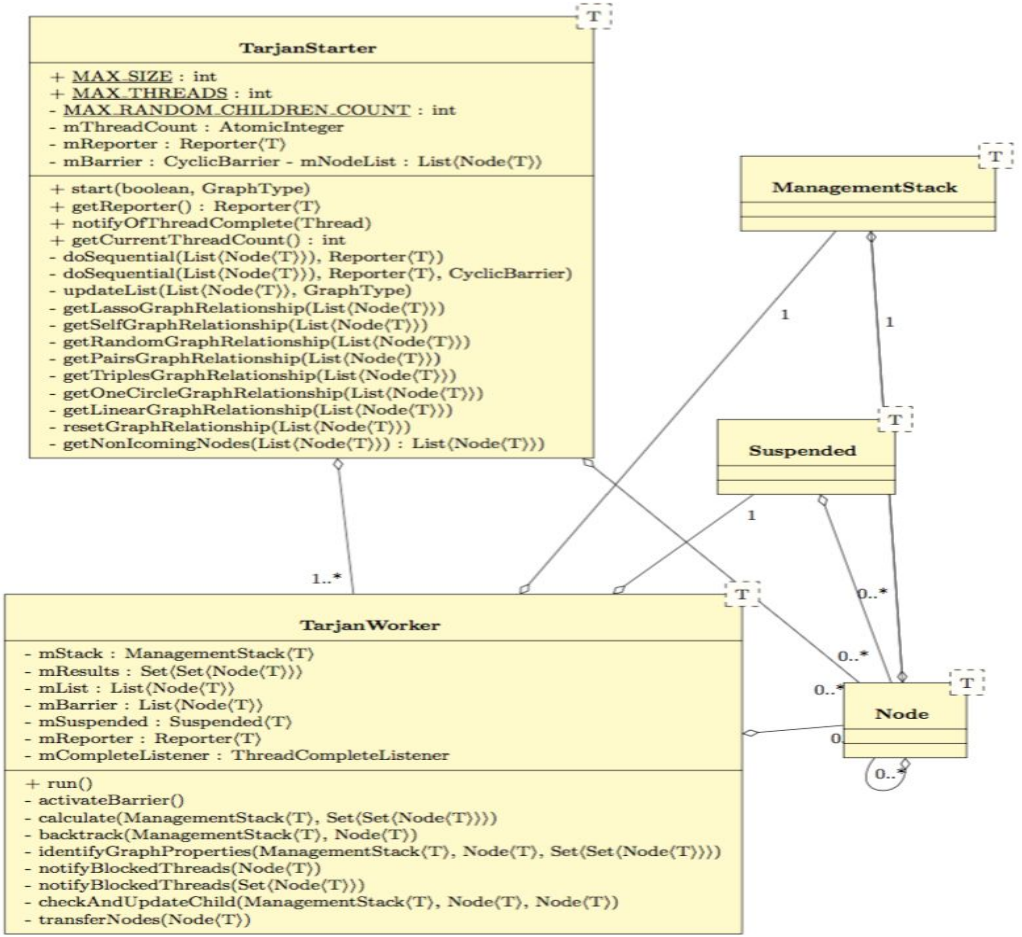
- Node
- ManagementStack
- Suspended



(a) Data Structures

Worker (Runnable)

- TarjanStarter
- TarjanWorker



(a) TarjanStarter & TarjanWorker

Pseudocode (1)

Initial node addition to a thread:

```
def addNode(node) = {
  node.index = index;
  node.lowlink = index;
  index += 1;
  node.search = thisSearch;
  controlStack.push(node);
  tarjanStack.push(node);
}
```

Pseudocode (2)

Main worker body:

```
while(controlStack.nonEmpty) {
  val node = controlStack.top;
  if (node has an unexplored edge to child) {
    if (child previously unseen) {
      addNode(child);
    } else if ( child is in tarjanStack ) {
      node.updateLowlink(child.index);
     } else if (child is not complete) {
      // child is inprogress in a different search
      suspend waiting for child to complete
    } else {
      // otherwise, child is complete, nothing to do
    }
  } else {
    // backtrack from node
    controlStack.pop;
    if(controlStack.nonEmpty) {
      controlStack.top.updateLowlink(node.lowlink);
      if(node.lowlink ==node.index) {
        start new SCC
        do {
          w = tarjanStack.pop;
          add w to SCC;
          mark w as complete and unblock any searches suspended on it
        } until (w == node);
      }
    }
  }
}
```

Transfer nodes on cyclic dependencies

- Get nodes to move.
- Update nodes.

```
private void transferNodes(Node<T> node) {
    assert(mSuspended.isThreadIdSuspended(node.getThreadId()));
    List<Node<T>> nodesToMove = node.getNodesToMove();
    int lIndex = nodesToMove.get(0).getLowlink();
    int maxIndex = lIndex;
    for (int index = 0; index < nodesToMove.size(); ++index) {
        Node<T> movedNode = nodesToMove.get(index);
        mStack.updateNodeTransfer(movedNode, lIndex);
        if (maxIndex < movedNode.getIndex()) {
            maxIndex = movedNode.getIndex();
        }
    }
    /*
     * Update stack index to max + 1 of the moved
     * nodes.
     */
    mStack.setStackIndex(maxIndex + 1);
}
```

Check Child

Process cycles & lassos.

```
private void checkAndUpdateChild(ManagementStack<T> stackManager,
        Node<T> node, Node<T> nextChild) {
    if (stackManager.addNode(nextChild)) {
        return;
    } else if (stackManager.tarjanStackContains(nextChild)) {
        node.updateLowlink(nextChild.getIndex());
        // Equal by reference  is good enough.
        if (node == nextChild) {
            node.markInCycle();
        }
        stackManager.markAllTarjanStackInLassos();
    } else {
        if (!nextChild.isComplete()) {
            if (mSuspended != null) {
                try {
                    mSuspended.lock();
                    if (mSuspended.isCyclicDependency(nextChild)) {
                        transferNodes(nextChild);
                    } else {
                        // A blocking call
                        mSuspended.suspendCurrentThreadWithBlock(nextChild);
                        if (nextChild.inLasso()) {
                            stackManager.markAllTarjanStackInLassos();
                        }
                    }
                } finally {
                    mSuspended.unlock();
                }
            }
        } else if (nextChild.inLasso()) {
            stackManager.markAllTarjanStackInLassos();
        }
    }
}
```

Node processing

- Process a node and mark it complete.
- notify for completed nodes.
- Process cycles & SCCs.

```
private void notifyBlockedThreads(Node<T> node) {
    if (mSuspended != null) {
        try {
            mSuspended.lock();
            mSuspended.signalAll(node);
        } finally {
            mSuspended.unlock();
        }
    }
}
```

```
private void identifyGraphProperties(ManagementStack<T> stackManager,
        Node<T> node, Set<Set<Node<T>>> results) {
    if (node.getLowlink() == node.getIndex()) {
        Set<Node<T>> set = new HashSet<Node<T>>();
        Node<T> w;
        do {
            w = stackManager.tarjanStackPop();
            set.add(w);
            w.markComplete();
        } while (w != node);
        results.add(set);
        if (set.size() > 1) {
            for (Node<T> cNode : set) {
                cNode.markInCycle();
            }
        }
        notifyBlockedThreads(set);
    }
}
```

Shared Memory Protection

- CyclicBarrier (single object, read only).
- **Suspended** (single object, read/write).
- Reporter (single object, write only).
- ThreadCompleteListener (single object, write only).
- **Node** (many objects, read/write).
- ManagementStack (one per thread, can be shared, read/write)

Protection

- Node.
- Suspended.
- During transfer nodes we sync(Node) as well.

```
while(controlStack.nonEmpty) {
  val node = controlStack.top;
  if (node has an unexplored edge to child) {
    if (child previously unseen) {
      addNode(child);
    } else if ( child is in tarjanStack ) {
      node.updateLowlink(child.index);
    } else if (child is not complete) {
      // child is inprogress in a different search
      suspend waiting for child to complete
    } else {
      // otherwise, child is complete, nothing to do
    }
  } else {
    // backtrack from node
    controlStack.pop;
    if(controlStack.nonEmpty) {
      controlStack.top.updateLowlink(node.lowlink);
      if(node.lowlink ==node.index) {
        start new SCC
        do {
          w = tarjanStack.pop;
          add w to SCC;
          mark w as complete and unblock any searches suspended on it
        } until (w == node);
      }
    }
  }
}
```

Sync(Node)

Sync(Suspended)

Sync(Suspended)

Testing - Graph Types

- SINGLE: a graph without any connection between nodes at all.
- PAIRS : a graph with a circular link between every two nodes sequentially
  - (node(i) ↔ node(i + 1)).
- ONE CIRCLE: a graph with a big circular link between all nodes sequentially
  - (node(0) → node(1), ..., node(i) → node(i + 1), node(i + 1) → node(i + 2), ..., node(n − 2) → node(n − 1), node(n − 1) → node(0)).
- SHARED PAIRS TRIPLES: a graph which has both PAIRS and TRIPLES features. This will allow the graph to have circles within circles which will end up as blocks of six with the following features
  - node(i) ↔ node(i + 1), node(i + 2) ↔ node(i + 3), node(i + 4) ↔ node(i + 5), && node(i) → node(i + 1), node(i + 1) → node(i + 2), node(i + 2) → node(i), node(i + 3) → node(i + 4), node(i + 4) → node(i + 5), node(i + 5) → node(i + 3)
- LASSO: a graph with a lasso repeated every ten nodes segment
  - node(i) → node(i+1),node(i+1) → node(i+2),...,node(i+8) → node(i+9),node(i+9) → node(i+7)
- RANDOM: this option generates random links between nodes with a random number of children per node. This was useful to detect interesting bugs during the development process.

Conclusions

- Concurrency is not easy.
- Testing is even harder.

Next steps?

- Test automation.
- Performance analysis.

Q&A

Thank you